[illegible]

[illegible]

[illegible]

[illegible]

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract No. F04701-85-C-0086-P00016 with the Space Systems Division, P.O. Box 92960, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by J. M. Straus, Acting Director, Computer Science Laboratory.

Lt. Chris Warack, SSD/CNDA was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) Program.

[illegible]
CHRISTOPHER WARACK, LT, USAF
MOIE Project Officer
SD/CNDA

[illegible]
JAMES A. BERES, LT COL, USAF
MOIE Program Manager
AFSTC/WCO OL-AB

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A because unclassified	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A because unclassified	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-0086A(2920-05)-2	5. MONITORING ORGANIZATION REPORT NUMBER(S) SD-TR-89-33
6a. NAME OF PERFORMING ORGANIZATION The Aerospace Corporation	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION Space Systems Division	
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 92957 Los Angeles, CA 90009-2957	7b. ADDRESS (City, State, and ZIP Code) Los Angeles Air Force Base Los Angeles, CA 90009-2960
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Space Division, Headquarters	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F04701-85-C-0086-P00016	
8c. ADDRESS (City, State, and ZIP Code) P.O. Box 92960 Los Angeles, CA 90009	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Gauge: Its Philosophy and Design (UNC)

12. PERSONAL AUTHOR(S)
Gorlick, Michael M., Kesselman, Carl F., Parker, D. Stott

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM ______ TO ______	1989 May 22	27

16. SUPPLEMENTAL NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
programming environments, software performance analysis

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Gauge is a workbench for gaining understanding of the performance of large parallel logic programs. A Gauge user is provided with a variety of interactive tools for analyzing the execution of programs. This report describes the philosophy, design, and implementation of Gauge and its use in analyzing parallel systems.

Keywords: Shared memory multiprocessors (KR) [illegible]

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED / UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Contents

1 Introduction

Understanding the performance properties of a large software system is problematic, particularly for parallel or distributed software systems. Several research fields in computer science are devoted to this concern, both to define performance or complexity measures and develope methodologies for determining whether programs meet these measures.

Logic programming is an interesting, but largely unexamined, platform for performance analysis. A promising formalism for writing parallel software, logic programs benefit from the clean semantics and performance potential of single-assignment languages [1]. Logic programs can also be viewed as data and are easily manipulated by analysis programs.

The importance of graphic environments that support system modeling and analysis is widely recognized [2, 3, 4, 5, 6, 7, 7, 8, 9]. Typically these environments provide the user with a trace facility coupled with tools for displaying the trace in a variety of useful ways.

Unfortunately, many of these environments encountered problems when applied as parallel-program analysis tools:

- Monitoring a parallel system (for example, with breakpoints and event tracing) may introduce overhead that perturbs program execution, to the extent that the execution of a traced system differs from that of the original.
- Standard tracing and breakpoint techniques for sequential programs are inadequate for parallel programs, and must be generalized to deal with a global state that may be difficult or impossible to determine, a nondeterminism of execution that may not be controllable, and execution behavior that may not be reproducible.
- Traces are a limited abstraction, and do not provide information corresponding to a user-level view or model of the system.

This report describes Gauge, an environment for measuring and analyzing the performance of parallel logic programs. Gauge is written in SICStus

Prolog and C, with graphics services supplied by the X Window System [10]. Although Gauge is intended for use with logic programs, we expect it will apply to other environments.

This report is organized as follows: After the philosophy of Gauge is made explicit in Section 2, Section 3 introduces concepts underlying the Gauge performance models. Section 4 then outlines the intended functionality and use of Gauge, and Section 5 summarizes the individual components of the system. Finally, Section 6 summarizes the contributions of Gauge.

2 Philosophy

Large pieces of industrial machinery have many dials and meters attached to them. In a normal plant environment, monitors regularly note the values from these gauges and record the status of the machines. Many inferences may be drawn from the values: for example, wear on turbine bearings can be determined from operating temperatures. Thus dangerous conditions and performance problems can be anticipated from the measurements.

Why is program monitoring not this straightforward? The instrumentation of programs is a painful, largely ad hoc process. Maintaining the performance history of programs is no better, and studying program performance in real time is usually impossible.

The goal of the Gauge environment can be summarized simply: to gain an understanding of the performance behavior of parallel logic programs running on shared-memory multiprocessors. This goal is related to many needs that are not addressed by existing environments today, specifically the following:

- Tools for the effective measurement of existing logic programs.
- Accurate general models of logic program performance.
- Useful tools for the user exploration of the execution behavior of logic programs.
- Techniques for the automated analysis of the execution behavior of logic programs.

What the performance analyst needs first is an *overall understanding* of the program. This understanding is critical, since it establishes the scale on which measurements are calibrated, the relative impact of components on the performance of the whole program, and the critical performance issues. Effective performance analysis requires perspective.

Simple models are best for getting a feel for the performance of a complex program. Sophisticated theories and deep models are usually not of much help, and do not necessarily lead to useful tools. A good analogy can be drawn from physics: although relativity theory has gradually become accepted as an accurate model of gravitation in the universe, in any earthly situation Newton's laws are a sufficient model.

We believe that performance analysis has its own sense of parsimony — most of the useful performance information comes from simple statistics that are inexpensive to gather. In performance analysis, less is more. Having complete information can be overwhelming, and it is expensive (or impossible) to gather in the first place. We can gain by understanding programs thoroughly in terms of simple models.

Though performance analysis benefits from its association with the precise formalisms of computational complexity and algorithmic analysis in the foreseeable future, neither discipline will have sufficient strength or breadth to encompass the performance questions of actual working systems [11, 12]. Rather than chase an elusive formal theory, we are better off admitting from the start that performance analysis is, and will be, a discipline based upon experimentation. The successful performance analysis environment will be a *skillful, mutually dependent, cooperative marriage of formal* analysis and experimentation.

With the points above in mind, Gauge emphasizes the following:

Low-Overhead Statistical Measurements While the overhead of tracing is usually high, certain run-time measurements can be made cheaply (for example, counts of entries and exits from predicate ports, maximum stack heights, and timing estimates [13]). Rather than seeking precise measurements at the cost of altering the very behaviors of interest, one should use nonintrusive statistical samples, accepting the indeterminacy of probe measurements as a basic fact of life in parallel processing. We believe that most *useful* performance information can be obtained from these low-overhead measurements.

Simple Analytic Models Just as Gauge is predicated on the thesis that much of the useful performance information comes from low-overhead measurements, it also is based on the belief that simple models of program execution provide most of the understanding about the program. Although performance measurements are easiest to obtain for logic program models (say, call graphs or proof trees), the models can apply to any level of abstraction. The point is that they be *simple*, so that they supply the overall program understanding we want. Models permitting *static analysis* are particularly important, since they can eliminate execution overhead.

Lifetime Monitoring Today, programs are usually stored separately from their documentation, and no historical information is kept about their evolution, use, or performance. This situation is unfortunate, since a great deal of useful information about programs can be found in their history. History also provides expectations about future evolution, use, and performance.

Gauge is designed so that performance information stays with a program over its entire lifetime. An advantage of low-overhead measurements is the possible storage of performance parameters with the program itself.

Some interesting issues in programming-environment technology are raised by these emphases. However, the philosophical thrust of Gauge is pragmatic — we simply want results.

3 Concepts

In this section we define the basic concepts behind Gauge. The Gauge vocabulary defines *programs*, *events*, *gauges*, and *execution models*.

In addition to maintaining information about a program over the program's lifetime, Gauge supports a number of *execution models*. Each model provides a different view of the program, and requires a different set of gauges. Aimed at low-cost/high-value performance modeling, Gauge supports simple models requiring only inexpensive measurements.

3.1 Programs

In its lifetime each program goes through a sequence of versions, and each version through a sequence of executions. Gauge maintains information about the program over the program's lifetime, since one can resolve many important performance questions only by examining this history. For example, program history describes the cumulative effects over time of changes in software, hardware configurations, and operating assumptions.

Programs go through a sequence of *versions* as they evolve. Thus whenever we talk about programs here, we mean versions of programs. Each program version can be characterized statically by the following information:

Predicates A program version corresponds to a collection of predicates, which have the following attributes:

1. **Definition** representing of the predicate that can be analyzed. Prolog predicate definitions consist of a set of *clauses*, each of which has a head, a body, and an abstract machine coding. Non-Prolog predicates have only trivial definitions that describe whether they are built-ins or external programs.
2. **Argument modes** specifying whether arguments are either input or output.
3. **Argument types** specifying the data types of arguments.
4. **Argument indexing** specifying how arguments are indexed.
5. **Parallelism assertions** suggesting possible *and*- and *or*-parallelism.
6. **Performance assertions** stating expected performance under expected operating conditions.

Performance assertions are one innovation encouraged by the lifetime monitoring of programs. When programmers develop programs, especially parallel programs, they make assumptions about operational conditions (throughput requirements, probable system loading, task allocation to available hardware, ratios of reads to writes for data structures, estimates of branching probabilities, and so forth). Today there is no means for recording these assumptions in a way that can be monitored automatically.

Static Call Graph The static call graph of a program version indicates which predicates call another. Although knowing how many times a predicate was called is part of the execution history, a complete picture comes only from knowing where the call came from.

Performance History A performance history is a sequence of results of executions. These results are the "long"-lived statistics kept after the execution of the program version, not the more ephemeral "short-lived" statistics gathered and displayed when the program version is run.

An execution consists of

1. **a machine configuration** showing exactly the hardware and software environment of the program;
2. **the commands used** showing exactly how the program was run; and
3. **the gauges available** specifying what performance information was kept.

Frequently, the commands used will specify a *test suite*, a set of controlled experiments.

3.2 Events

During an execution a change in any program object is viewed as an *event*. Gauge represents events at three levels:

Abstract Machine-Level Events The abstract machine level is the lowest level represented by Gauge, capturing the operational behavior of the Prolog emulator that executes the abstract machine instructions produced by the compiler. This operation includes *instruction events* that are raised when individual instructions are encountered by the emulator, and *storage events*, such as abstract machine stack (environment, trail, heap) changes, register use, and so forth.

Logic Program-Level Events One level up, at the program level, events are naturally divided into two types. *Predicate events* record the port entry and exit of every predicate, the selection of clauses, and so forth. *Call-graph events* are raised when one predicate calls another (or itself).

Program Model-Level Events Programmers conceptualize programs above the abstract machine level and the logic program level, reflecting the domain of the problem solved by the program. For example, a program simulating a satellite might be modeled in terms of the functions of the satellite. Events are determined by the model under consideration; an event in satellite simulations could be the rotation of the satellite's solar panels in response to an attitude change. Gauge supports the development of simple program models.

Events have the following attributes:

Type Events are either *boolean* (they happen), or *valued* (they involve some change of value).

Display Information Associated with each type of event is information about how that event may be most effectively displayed. This is important in displaying gauges based upon the event. A format for encoding this information is described in [5, 6].

3.3 Gauges

A *gauge* is a value or measurement extracted from a program execution in real time. Measurements in Gauge are restricted to have low overhead – requiring only minimal computational effort to gather. This reduces the impact of monitoring on execution, and eliminates the need for expensive synchronization techniques, such as locking, in saving parallel-processing measurements.

Internal Gauges An internal gauge describes the occurrences of some program event over time, and is updated by the program itself. Since the program can be affected by any significant effort spent in updating these gauges, the effort is restricted to require at most a few machine instructions.

Recall that events are either boolean (they happen), or valued (they involve some change of value). An example of a boolean event is a program reaching a certain predicate, while valued events include stack pointer changes and selections of clauses.

The only gauges available for boolean events are *counts*.

For valued events the following internal gauges are available.

Constant-Time Statistics A constant-time statistic is a numeric value that can be gathered easily from valued events. Gauge supports statistics such as

- **minimum**
- **maximum**
- **average** (running count and sum)

- **variance** (running count, sum, and sum of squares)

Bounded Samples Frequently in large systems code it is useful to monitor windows of activity on some valued event. For example, we might want to record the identifiers of the last ten clauses invoked, or identifiers of the ten most frequently invoked clauses. This can be done by storing samples of some variable in a circular buffer of size n. Gauge supports the following bounded samples

- **last** n
- **first** n
- **maximum** n
- **minimum** n
- n **quantile estimates**

The time to perform this storage is usually constant, needing only a few machine instructions. Fast algorithms are used for quantile estimation [14, 15].

External Gauges All the internal gauges mentioned above can be inspected by some monitor process external to the program itself. On a shared-memory multiprocessor, this inspection can be done without a serious impact on performance. In addition, some gauges involving more global information are useful and are available with Gauge. Perhaps the most important example is the *snapshot*.

External gauges operate through monitor processes that either run in parallel with, or stop, the program being measured. These monitor processes obtain gauges by inspecting the program state. Thus whatever program state information is needed for external observation must be made visible.

The basic parameter determining the accuracy of external gauges is their *sampling rate*. The sampling rate is a tunable value; by increasing it we obtain more information, although a higher level of accuracy is achieved at the cost of increased bandwidth and storage requirements. Thus the internal gauges above can also be obtained externally, but statistically rather than precisely. However, by increasing the sampling rate to approach the clock speed of the processor running the program being monitored, we can obtain, as a limit, complete traces of the program. The performance analyst can trade off the expense of keeping more timely statistics with the accuracy they offer.

There are three important kinds of external gauges:

timing gauges Internal gauges cannot afford the overhead implied by real-time clock information. Thus all timing information concerning events (for example, inter-event time, average path execution times, and frequencies) are obtained by external gauges.

snapshots A snapshot saves a timestamped copy of the internal gauges of the program being monitored. Obtaining this global information does not necessarily require stopping the program. A background process can run in parallel with (but on a different processor than) the program and gather the information needed.

derived gauges Gauges may be either *primitive* (directly implemented at the machine level) or *derived*. A derived gauge combines several primitive internal gauges into a more abstract value. Derived gauges can be useful in capturing program model events.

A sequence of snapshots $S_1, ..., S_n$ is called a *history*. Note that Gauge does not include trigger-type gauges, which keep track of multiple events occurring in prespecified ways. Although it is conceivable that these gauges could be implemented efficiently as external gauges (using, for example, the RETE-net technology of modern production systems such as OPS-5), they are counter to the philosophy of Gauge. Higher-level abstractions of event trace information can be extracted *post facto* by individual execution-model performance-analysis systems.

A gauge has several attributes:

Event Type Internal gauges and external timing gauges correspond to certain program events.

Lifetime Gauges can be *long-lived* or *short-lived*, depending on whether the result is saved after the execution or is discarded (having been used only for interactive, real-time display). Long-lived gauges tend to be associated directly with program structures, and are time-insensitive in that they are computed only once, or are averaged over time. Short-lived measurements tend to be associated with collections of program objects, or consist of the values of a single program structure sampled over time.

3.4 Execution Models

An execution model is an abstraction of the available gauge values, and is usually presented at the program level. If the history of gauge values is a data base, the execution model is a view of this data base.

At a minimum, we are interested in looking at three basic aspects of programs: the *timing* of individual components, the *flow* patterns among individual components, and the overall use of *space*. Our program models must provide perspective on these aspects.

Gauge supports the following simple models of program structure and execution:

Abstract Machine Model Logic programs are usually compiled into instructions for some variant of the well-known Warren Abstract Machine (WAM) [16]. Events in this model correspond to state changes in the abstract machine: instruction sequencing, register contents, stack contents, and so forth.

Dynamic Call Graph Model The static call graph stored with every program reflects who calls whom. The dynamic call graph uses the same structure, but decorates it with frequencies: who calls whom *how often*. In addition, the dynamic call graph reflects invocations of predicates via higher-order primitives such as **call/1** and **setof/3**, which may not be included in the static call graph.

Dynamic Proof Tree Model The execution of every logic program can be thought of as a process of constructing a tree. In a parallel context, this is viewed as the "parallel painter model."

Statistical Model Statistical models represent a program as a collection of independently sampled random variables. An example is the timing model of [13].

Other abstract models of programs will be supported as the need arises, including, for example, logical ripple-effect models, control flow models, phase models (which divide a program into logical phases or parts), extended queuing network models, Petri net models, and so forth.

4 Functions

Like most empirical parts of computer science, performance evaluation is an iterative affair. Gauge supports three phases of performance evaluation, reflecting our opinions about the methodology that performance analysts would apply to real Prolog programs. These phases are static performance analysis, execution design, and execution performance analysis.

Initially, Gauge performs a static analysis of programs for potential performance problems. This analysis provides several functions, including a mechanism to warn about poor performance, a source of information enabling the program developer to improve programs, and a source of ideas for designing experiments to test the program. Just as software developers first rely on tools to perform "lint checking" to eliminate possibly dangerous programming techniques from their programs, static analysis highlights performance problems.

Once the program appears ready to run, the first goal of the performance analyst is to get a "feel" for the program's behavior. This is done by repeatedly designing suitable test suites for the program and then analyzing the resulting data obtained from low-overhead program models. Roughly speaking, the idea is to "search and destroy bottlenecks." The search thus inspects only a small portion of the program's performance behavior.

4.1 Static Performance Analysis

The static analysis of programs [17] has two main benefits: it provides the programmer information about the program, and it can be used to increase the efficiency of program execution. With flow analysis, for example, one can remove useless or redundant code, rearrange goals, and improve partial evaluation. This information is important in shortening the performance improvement cycle. By avoiding obvious pitfalls early, static analysis speeds overall development.

It seems that static analysis can also serve as a manager of heuristic information about performance. User-specified performance assertions and program annotations describing likely parallelism in program clauses can be useful in static analysis. Static analysis can serve as a repository of common wisdom about program performance problems. Many performance problems arise repeatedly in programs written by less-expert programmers and in programs generated by combinations of tools (such as programs generated by a sequence of source-to-source transformations).

These problems are not really *errors* of programming - they are simply pitfalls, or poor-performance-prone ways of coding. Static analysis can therefore alert the programmer about potential performance issues in his program.

Static analysis permits us to answer performance-related questions such as the following:

- Does the program make heavy use of the external program interface?
- Which clauses appear to manifest bad coding style? Bad Prolog coding style is difficult to make precise, but the following guidelines illustrate the approach taken by Gauge:
  - superfluous cuts
  - inefficient use of ==/2 or =/2
  - the unnecessary use of known expensive predicates such as **assert**, **retract**, or =..
  - large terms in the heads of clauses
  - long clauses
  - disjunctions in clause bodies

In addition to these, static analysis includes queries that an abstract-machine compiler can immediately use in improving code:

- Which recursive predicates are not tail recursive?
- Would changing the argument order reduce the number of WAM registers used?
- Would changing the subgoal order or eliminating disjuncts in clause bodies improve the WAM code generated?

- Are there global variables that can be eliminated? Are unnecessary structures created?
- Where could recursion be advantageously replaced by repeat-fail?

Thus static analysis covers many of the queries we would expect of a powerful optimizing compiler.

Static analysis goes beyond compilation techniques, however. When specific branching probabilities are affixed to edges of the call graph, program performance can be estimated through simulation. This simulated execution is an example of abstract interpretation [18, 19, 20, 21], in which one estimates the behavior of a program by simulating some abstracted version of it. This approach underlies most static program-analysis techniques [19].

4.2 Execution Design

The Gauge measurement harness permits users to specify the runs to be made, and for which short-lived models, if any, the runs should make measurements.

Two kinds of executions can be specified:

1. **specific execution** A specific goal (or sequence of goals) is to be run.
2. **test suite** Test suites generated from underspecified Mockingbird constraints [22] can be *explicit* (constraints made specific by test requirements), *randomized* (constraints made specific by random assignment), or *goal-oriented.* A goal-oriented suite is an executive that repeatedly executes the program either in a hill-climbing attempt to maximize some parameter, or in a randomized way, seeking some specific behavior such as the occurrence of an event.

The Mockingbird system [22] is used as a way to specify which gauges are to be enabled in a test suite, and provides a declarative way to state what the execution should accomplish. Performance constraints are specified as in [23].

4.3 Execution Performance Analysis

Imagine now that our program is finally running on a multiprocessor. What displays do we want to see? What queries must we ask to get a feeling for the program's behavior? Although no two people will view exactly the same things as important, some common queries include the following:

- Which predicates are called most often?
- Which subtree of the program call graph uses the most cpu time?
- Which specific predicate calls wind up taking most of the cpu time over all calls to that predicate? (How long did **q** take whenever it was called by **p**?)
- Ignoring the predicates that, while time-consuming, cannot be avoided, which predicates look like the bottlenecks?
- Which predicates create unnecessary choice points?
- Which clauses contribute most to stack depth? By stack we mean any of the environment, heap, trail, and choice point stacks, so answering this query can require significant amounts of data processing.
- Which clauses regularly do lots of work before failing?
- What would improve the failure profile (changing the argument order, indexing, breaking the clause up into smaller pieces)?
- Which clauses are active during (and hence possibly cause) garbage collection?
- Are lists used where functors (vectors, arrays) might be faster?
- Is the interpreter being used unnecessarily?
- What is the success profile of unifications against arguments?
- How does the profile suggest arguments should be reordered?
- What is the success profile of the built-in type-testing primitives such as **var/1** or **integer/1**?
- What is the success profile of unifications against clauses?

- How does the profile suggest clauses should be reordered?

Here is the point: *all of these queries ask about the behavior of simple analytical program models, and can be answered from low-level measurements.* More than this is rarely needed: once we have answers to these queries, it is usually evident what parts of the program need attention, and more or less how they should be changed.

Gauge offers limited event gathering and limited viewing of the events. It does not permit programmers to generate arbitrary event data bases and ask general temporal queries. Events can be gathered only for any predefined program model, and can be analyzed only by predefined execution-analysis tools. Since Gauge is extensible, new models and analysis tools can be added as needed, without any real loss in power.

Queries can be initiated in one of the two following ways:

1. **static analysis corroboration** The potential performance hotspots identified by static analysis can be automatically investigated when real execution performance data are available. In addition, branching probabilities obtained from execution can be used to improve static program models. This information is of great importance in program partitioning and task allocation.
2. **user queries** Naturally, interactive query by users is the main way a feel for the program's behavior can be obtained. One benefit of the limited model approach of Gauge is that common performance queries can be organized in menus for easy access, avoiding the clumsiness of query languages.

The queries listed above are basic and apply to both sequential and parallel environments. For parallel environments, however, much more sophisticated queries are needed. We must be able to evaluate the relationship between program events and memory contention, I/O contention, cache conflicts, load imbalance (poor task allocation), the improper use of bag predicates or other multiple-solution primitives, and the general use of parallel operating system primitives. Gauge provides support for all of these.

5 Subsystems

Gauge incorporates the following different components.

5.1 Static Performance Analysis

The static analyzer performs extensive global analysis of programs, and includes a performance "lint checker" for programs decorated with performance assertions and other programmer annotations. The output of the analysis includes performance warnings to the programmer, program model information used by the Prolog compiler, program model information used in execution performance analysis, and tests to be run after execution to verify conjectured performance hotspots.

5.2 Execution Harness

Gauge has an environment for creating test suites, assigning processes to processors, initializing and running the program, and capturing results in an event data base.

5.3 Event Data Base

The event data base is a fast, sequential storage manager with some support for time series analysis. A full-function data base manager and query processor is not necessary here, since the performance analysis problem is so special: the data base is append-only, and is read-only after execution; functionally the emphasis is on very rapid insertion and moderately rapid sequential scanning. Also, these data bases are used only briefly, and most queries are run only once.

The event data base is updated either during program execution by external gauges, or after execution terminates and internal gauges of the program are captured.

5.4 Presentation Functions and Execution Analysis

Since graphics are essential to a subjective understanding of numerical data, Gauge provides a variety of different ways of viewing these data: static presentations (plots), dynamic presentations (movies), browsing, and statistical analysis.

Static presentations can be either textual or graphic. Graphic presentations give plots of specific variables of interest; conventional curve, surface, and histogram plotting; gestalt-like plots; and other displays that maximize information transfer [5, 6, 24].

Dynamic presentations also can be textual or graphic, and include animation, moving gauges/dials, and model-oriented movies. Postmortem displays can be rerun, can use different timing speeds, and so forth. This permits one to browse through subhistories of interest, abstract subhistories (zoom in/out), and view a program from the perspective of multiple models simultaneously.

Some models require statistical analysis and the display of performance data. Gauge automates some of this analysis, such as linear-regression analysis and general curve fitting. These combine to yield a semi-automated complexity analysis. While this does not provide a complete model-validation mechanism, it does provide the means for making comparisons between expected and actual complexity.

6 Conclusion

Gauge gives a system designer a practical tool for investigating large software system behavior during (parallel) execution. Its approach emphasizes the use of *low-level mechanisms* of low cost and only statistical accuracy. These mechanisms have the benefit that they are nonintrusive, yet they extract most of the useful information about execution behavior.

Gauge also emphasizes the use of a limited number of *program models* that can be displayed or analyzed in predefined ways, as well as the *lifetime monitoring* of programs to obtain performance histories that are unfortunately not preserved today. The system is extensible and will evolve to support the tools that are most useful in program performance analysis.

References

[1] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, (MIT Press, 1986).

[2] W. Gale, *Artificial Intelligence and Statistics*, (Addison-Wesley, 1986).

[3] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems," *ACM Trans. Computer Systems*, Vol. 5, pp. 121–150, May 1987.

[4] J. Kurose, K. Gordon, R. Gordon, E. MacNair, and P. Welch, "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," in *IEEE Fall Joint Computer Conf. (FJCC-86)*, pp. 719–728, 1986.

[5] J. Mackinlay, "Automatic Design of Graphical Presentations," Report STAN-CS-86-1138, Dept. of Computer Science, Stanford University, December 1986.

[6] J. Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics*, Vol. 5, pp. 110–141, April 1986.

[7] B. Melamed, "The Performance Analysis Workstation: An Interactive Animated Simulation Package for Queueing Networks," in *IEEE Fall Joint Computer Conf. (FJCC-86)*, pp. 729–740, 1986.

[8] J. Sinclair and S. Madala, "A Graphical Interface for Specification of Extended Queueing Network Models," in *IEEE Fall Joint Computer Conf. (FJCC-86)*, pp. 709–718, 1986.

[9] R. Thisted, "Computing Environments for Data Analysis," *Statistical Science*, Vol. 1, no. 2, pp. 259–275, 1986.

[10] R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, Vol. 5, pp. 79–109, April 1986.

[11] J. Cohen, "Computer-Assisted Microanalysis of Programs," *Comm. ACM*, Vol. 25, pp. 724–733, October 1982.

[12] L. Ramshaw, "Formalizing the Analysis of Algorithms," Technical Report 79-741, Computer Science Dept., Stanford University, June 1979.

[13] M. Gorlick and C. Kesselman, "Timing Prolog Programs Without Clocks," in *Proceedings Fourth Symposium on Logic Programming*, pp. 426–432, IEEE Computer Society, 1987.

[14] R. Jain and I. Chlamtac, "The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations," *Communications ACM*, Vol. 28, no. 10, pp. 1076–1085, 1985.

[15] J. Pearl, "A Space-Efficient On-Line Method of Computing Quantile Estimates," *Journal. of Algorithms*, Vol. 2, pp. 164–177, 1981.

[16] D. H. D. Warren, "An Abstract Prolog Instruction Set," technical note, SRI International, Artificial Intelligence Center, October 1983.

[17] M. Hecht, *Flow Analysis of Computer Programs*, (Elsevier North-Holland, 1977).

[18] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," in *Proceedings Fourth Symposium on Logic Programming*, pp. 192–204, IEEE Computer Society, 1987.

[19] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Fourth Symposium on Principles of Programming Languages*, pp. 238–252, January 1977.

[20] S. Debray and D. Warren, "Automatic Mode Inference for Prolog Programs," in *Proceedings Third Symposium on Logic Programming*, pp. 78–88, IEEE Computer Society, 1986.

[21] C. Mellish, "Abstract Interpretation of Prolog Programs," in *Proceedings Third International Conf. on Logic Programming*, pp. 463–474, Springer-Verlag, 1986.

[22] M. Gorlick, C. Kesselman, D. Marotta, and D. Parker, "Mockingbird: A Logical Methodology for Testing," Technical Report ATR-86A-(8544)-3, The Aerospace Corporation, June 1987.

[23] R. McCartney, "Synthesizing Algorithms with Performance Constraints," in *Proceedings AAAI 1987*, pp. 149–154, July 1987.

[24] E. R. Tufte, *The Visual Display of Quantitative Information*, (Graphics Press, 1983).

LABORATORY OPERATIONS

The Aerospace Corporation functions as an "architect-engineer" for national security projects, specializing in advanced military space systems. Providing research support, the corporation's Laboratory Operations conducts experimental and theoretical investigations that focus on the application of scientific and technical advances to such systems. Vital to the success of these investigations is the technical staff's wide-ranging expertise and its ability to stay current with new developments. This expertise is enhanced by a research program aimed at dealing with the many problems associated with rapidly evolving space systems. Contributing their capabilities to the research effort are these individual laboratories:

Aerophysics Laboratory: Launch vehicle and reentry fluid mechanics, heat transfer and flight dynamics; chemical and electric propulsion, propellant chemistry, chemical dynamics, environmental chemistry, trace detection; spacecraft structural mechanics, contamination, thermal and structural control; high temperature thermomechanics, gas kinetics and radiation; cw and pulsed chemical and excimer laser development including chemical kinetics, spectroscopy, optical resonators, beam control, atmospheric propagation, laser effects and countermeasures.

Chemistry and Physics Laboratory: Atmospheric chemical reactions, atmospheric optics, light scattering, state-specific chemical reactions and radiative signatures of missile plumes, sensor out-of-field-of-view rejection, applied laser spectroscopy, laser chemistry, laser optoelectronics, solar cell physics, battery electrochemistry, space vacuum and radiation effects on materials, lubrication and surface phenomena, thermionic emission, photo-sensitive materials and detectors, atomic frequency standards, and environmental chemistry.

Computer Science Laboratory: Program verification, program translation, performance-sensitive system design, distributed architectures for spaceborne computers, fault-tolerant computer systems, artificial intelligence, micro-electronics applications, communication protocols, and computer security.

Electronics Research Laboratory: Microelectronics, solid-state device physics, compound semiconductors, radiation hardening; electro-optics, quantum electronics, solid-state lasers, optical propagation and communications; microwave semiconductor devices, microwave/millimeter wave measurements, diagnostics and radiometry, microwave/millimeter wave thermionic devices; atomic time and frequency standards; antennas, rf systems, electromagnetic propagation phenomena, space communication systems.

Materials Sciences Laboratory: Development of new materials: metals, alloys, ceramics, polymers and their composites, and new forms of carbon; non-destructive evaluation, component failure analysis and reliability; fracture mechanics and stress corrosion; analysis and evaluation of materials at cryogenic and elevated temperatures as well as in space and enemy-induced environments.

Space Sciences Laboratory: Magnetospheric, auroral and cosmic ray physics, wave-particle interactions, magnetospheric plasma waves; atmospheric and ionospheric physics, density and composition of the upper atmosphere, remote sensing using atmospheric radiation; solar physics, infrared astronomy, infrared signature analysis; effects of solar activity, magnetic storms and nuclear explosions on the earth's atmosphere, ionosphere and magnetosphere; effects of electromagnetic and particulate radiations on space systems; space instrumentation.